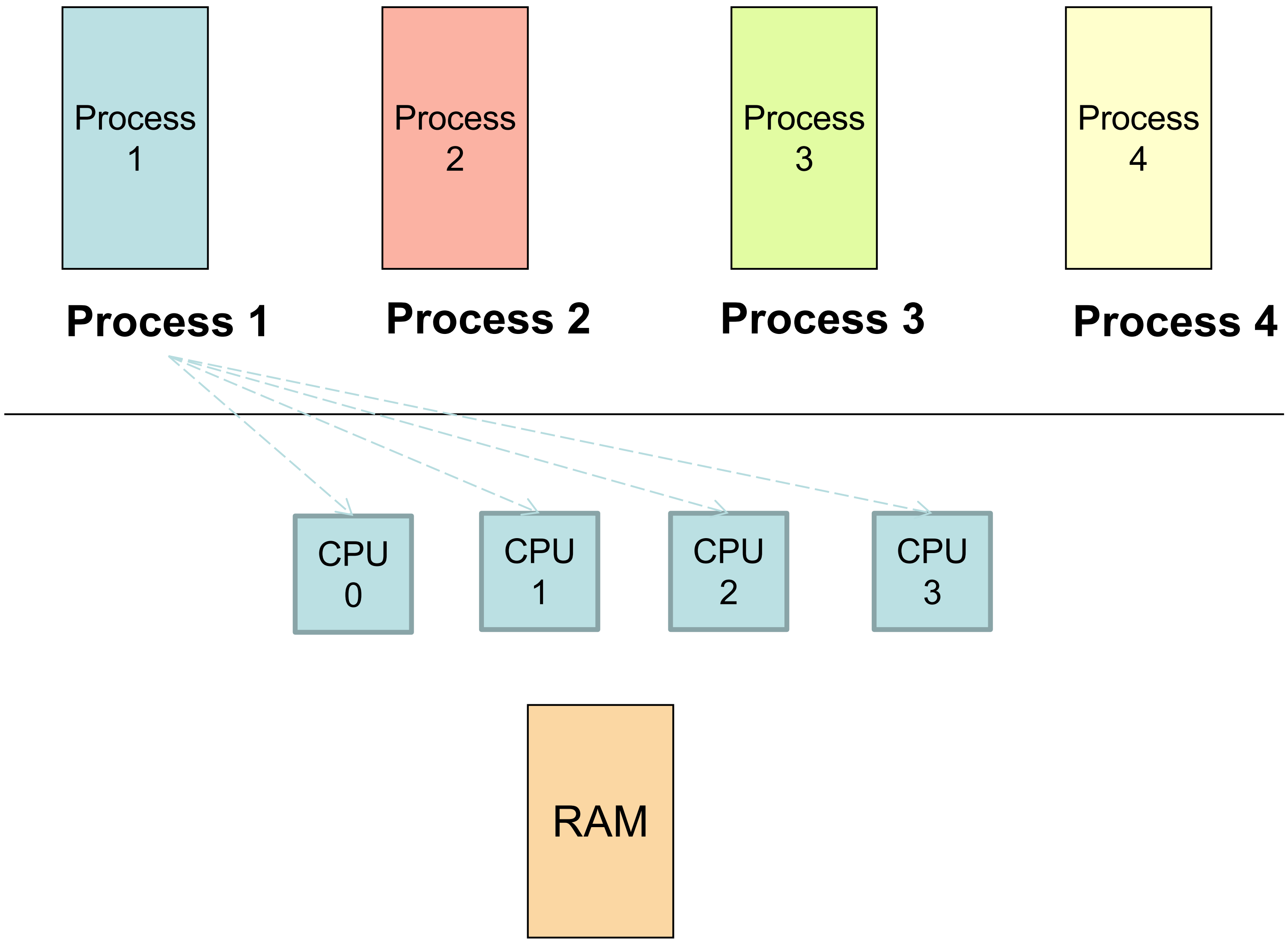
CPU Scheduling

Chester Rebeiro
IIT Madras

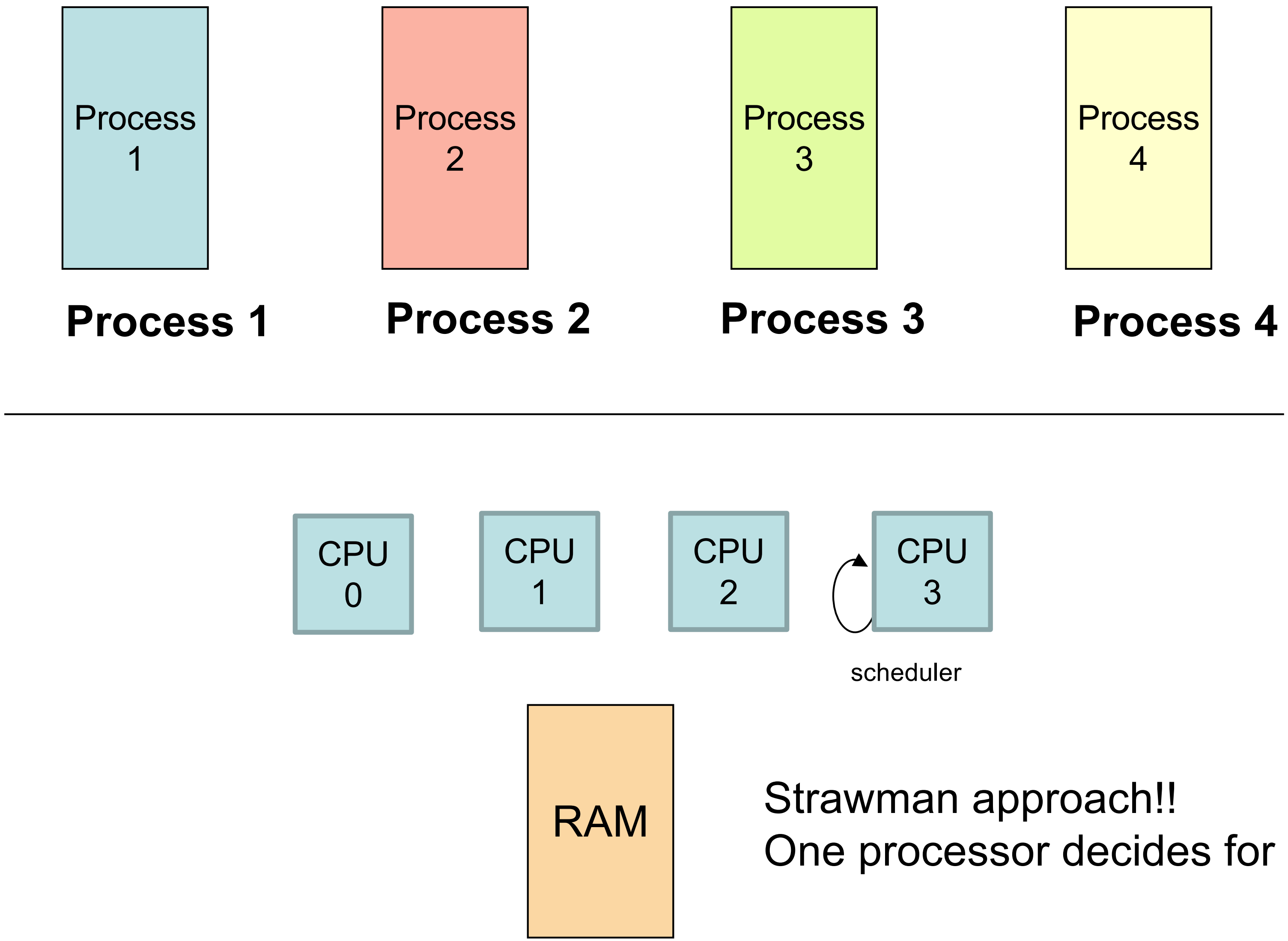
Multiprocessor Scheduling



Process Migration

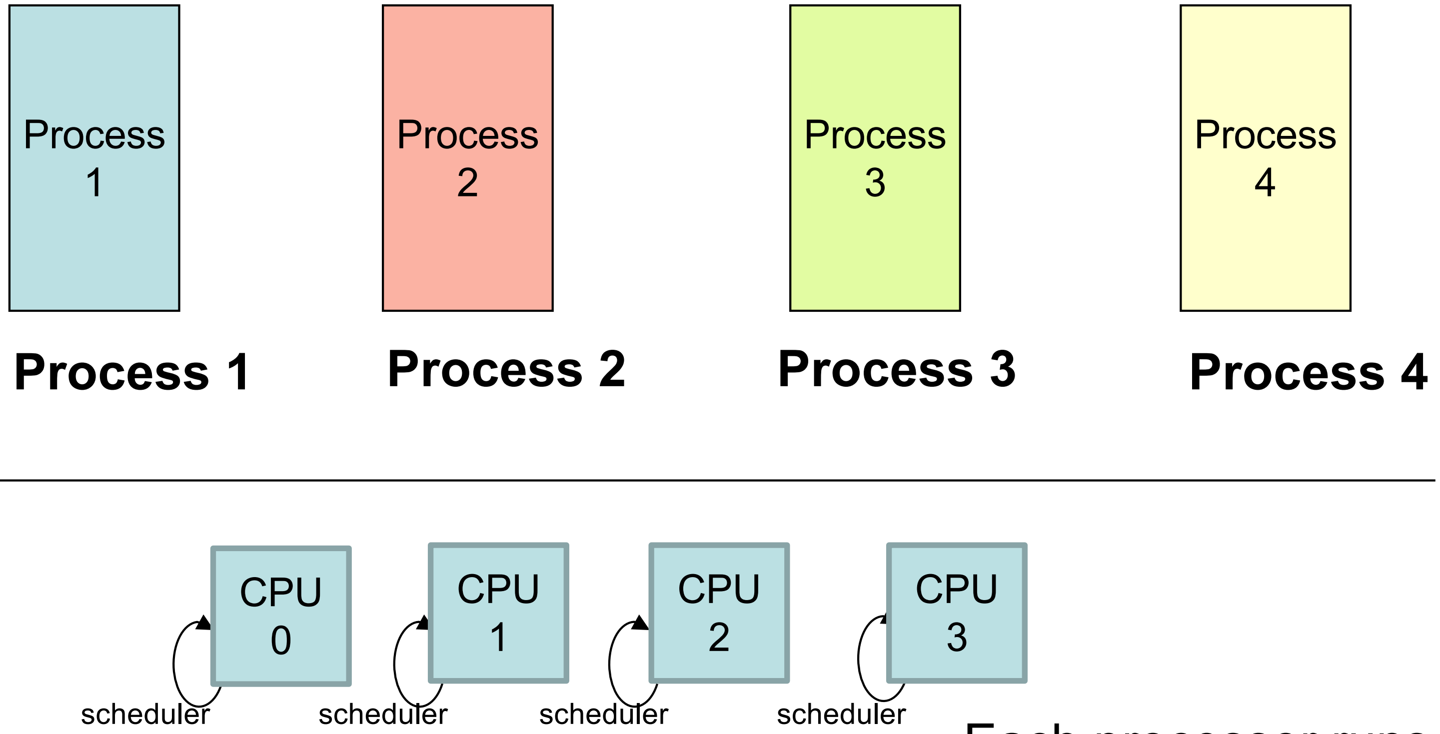
- As a result of symmetrical multiprocessing
  - A process may execute in a processor in one timeslice and another processor in the next time slice
  - This leads to process migration
    - Migration is expensive, it requires all memories to be repopulated
- Processor affinity
  - Process has a bitmask that tells what processors it can run on
    - Two types of processor affinity
      - Hard affinity – strict affinity to specific processors
      - Soft affinity

Multiprocessor Scheduling with a single scheduler



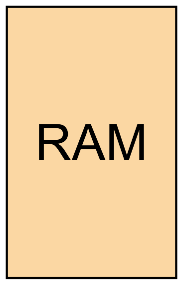
Strawman approach!!
One processor decides for everyone

Multiprocessor Scheduling (Symmetical Scheduling)



Each processor runs a scheduler independently to select the process to execute

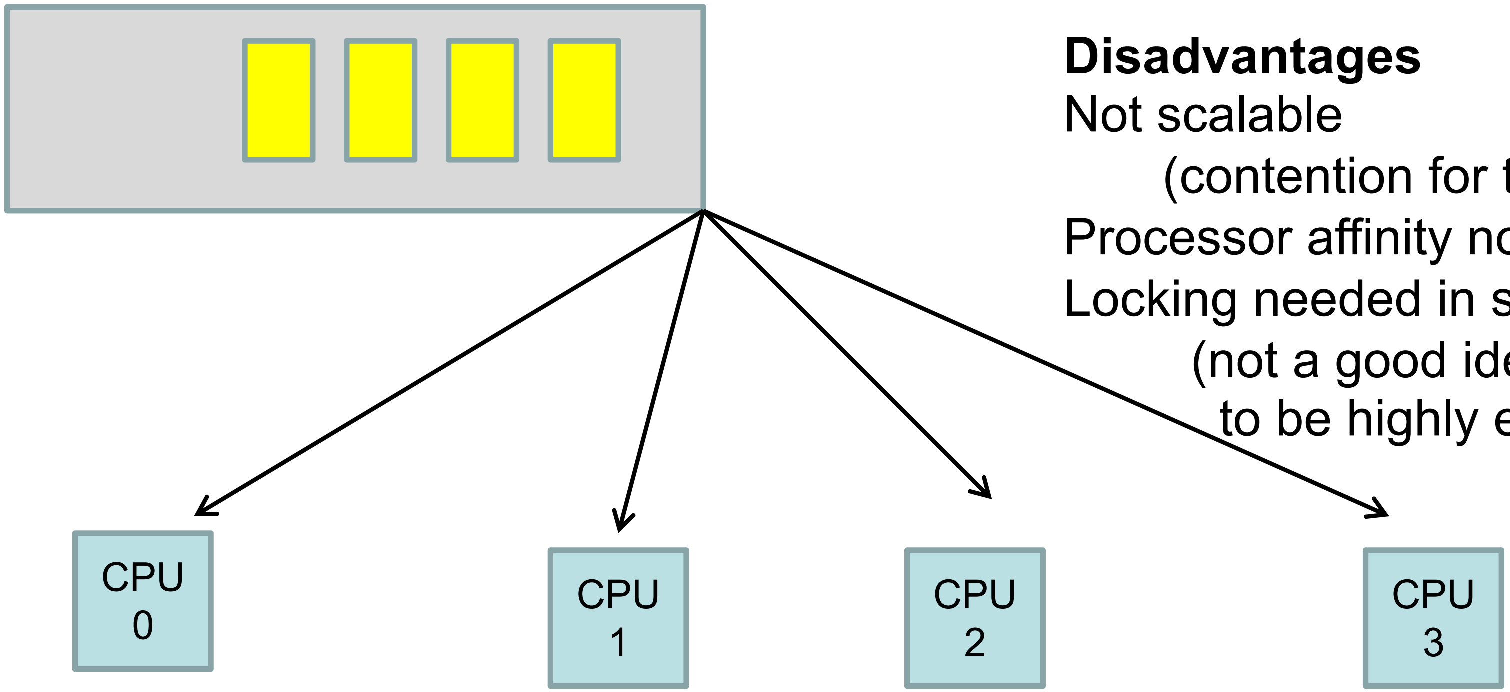
Two variants,
- Global queues
- Per CPU queues



Requires locking to access the queues

Symmetrical Scheduling (with global queues)

Global queues of runnable processes



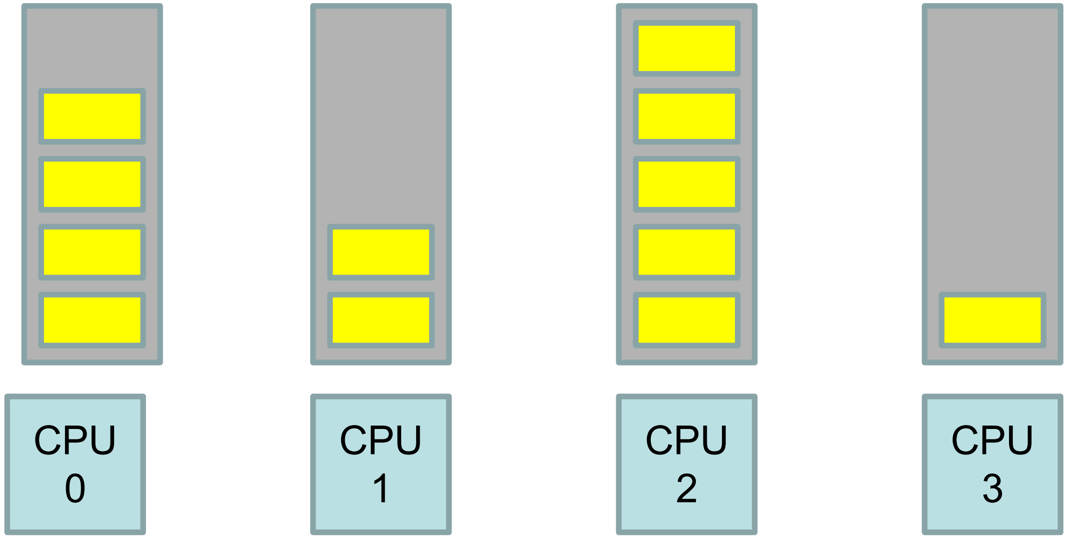
Advantages
Good CPU Utilization
Fair to all processes

Disadvantages
Not scalable
(contention for the global queue)
Processor affinity not easily achieved
Locking needed in scheduler
(not a good idea. Schedulers need to be highly efficient)

Used in Linux 2.4, xv6

Symmetrical Scheduling (with per CPU queues)

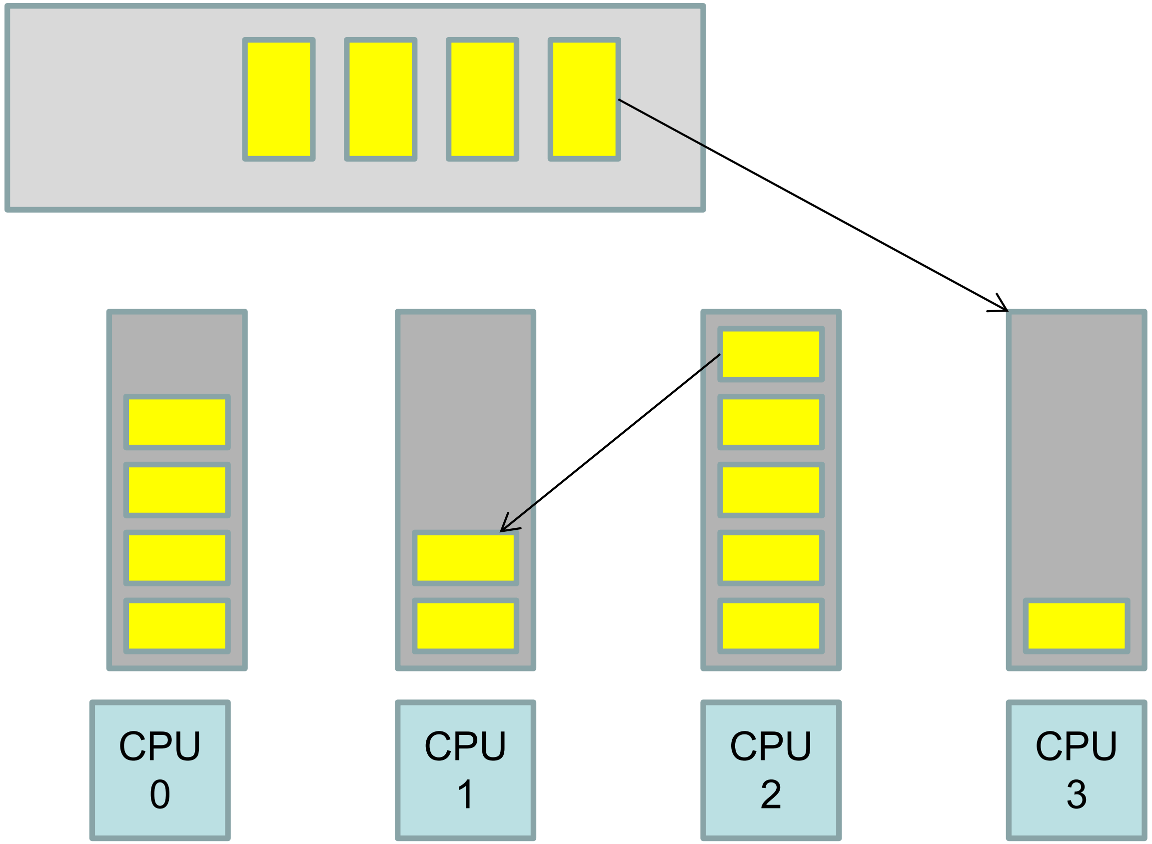
- Static partition of processes across CPUs



Advantages
Easy to implement
Scalable (no contention)
Locality

Disadvantages
Load imbalance

Hybrid Approach



- Use local and global queues
- Load balancing across queues feasible
- Locality achieved by processor affinity wrt the local queues
- Similar approach followed in Linux 2.6

Load Balancing

- On SMP systems, one processor may be overworked, while another underworked
- Load balancing attempts to keep the workload evenly distributed across all processors
- Two techniques
  - Push Migration : A special task periodically monitors load of all processors, and redistributes work when it finds an imbalance
  - Pull Migration : Idle processors pull a waiting task from a busy processor

Scheduling in Linux

Chester Rebeiro
IIT Madras

Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel,* 3rd Edition

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low priority task
- Normal Processes
  - Interactive
    - Constantly interact with their users, therefore spend a lot of time waiting for key presses and mouse operations.
    - When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)
  - Batch
    - Does not require any user interaction, often runs in the background.

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low
- Normal Processes
  - Interactive
    - Constantly interact with their users, therefore spend a lot of time waiting for key presses and mouse operations.
    - When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)
  - Batch
    - Do not require any user interaction, often run in the background.

Once a process is specified real time, it is always considered a real time process

Process Types

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low
- Normal Processes
  - Interactive
    - Constantly interact with their users key presses and mouse operations
    - When input is received, the proces between 50 to 150 ms)
  - Batch
    - Do not require any user interaction, often run in the background.

A process may act as an interactive process for some time and then become a batch process.

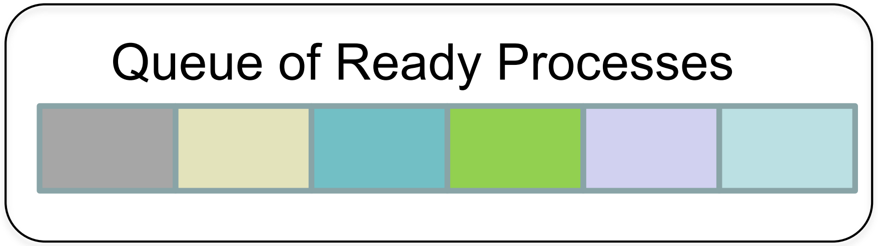
Linux uses sophisticated heuristics based on past behavior of the process to decide whether a given process should be considered interactive or batch

History (Schedulers for Normal Processors)

- O(n) scheduler
  - Linux 2.4 to 2.6
- O(1) scheduler
  - Linux 2.6 to 2.6.22
- CFS scheduler
  - Linux 2.6.23 onwards

O(n) Scheduler

- At every context switch
  - Scan the list of runnable processes
  - Compute priorities
  - Select the best process to run



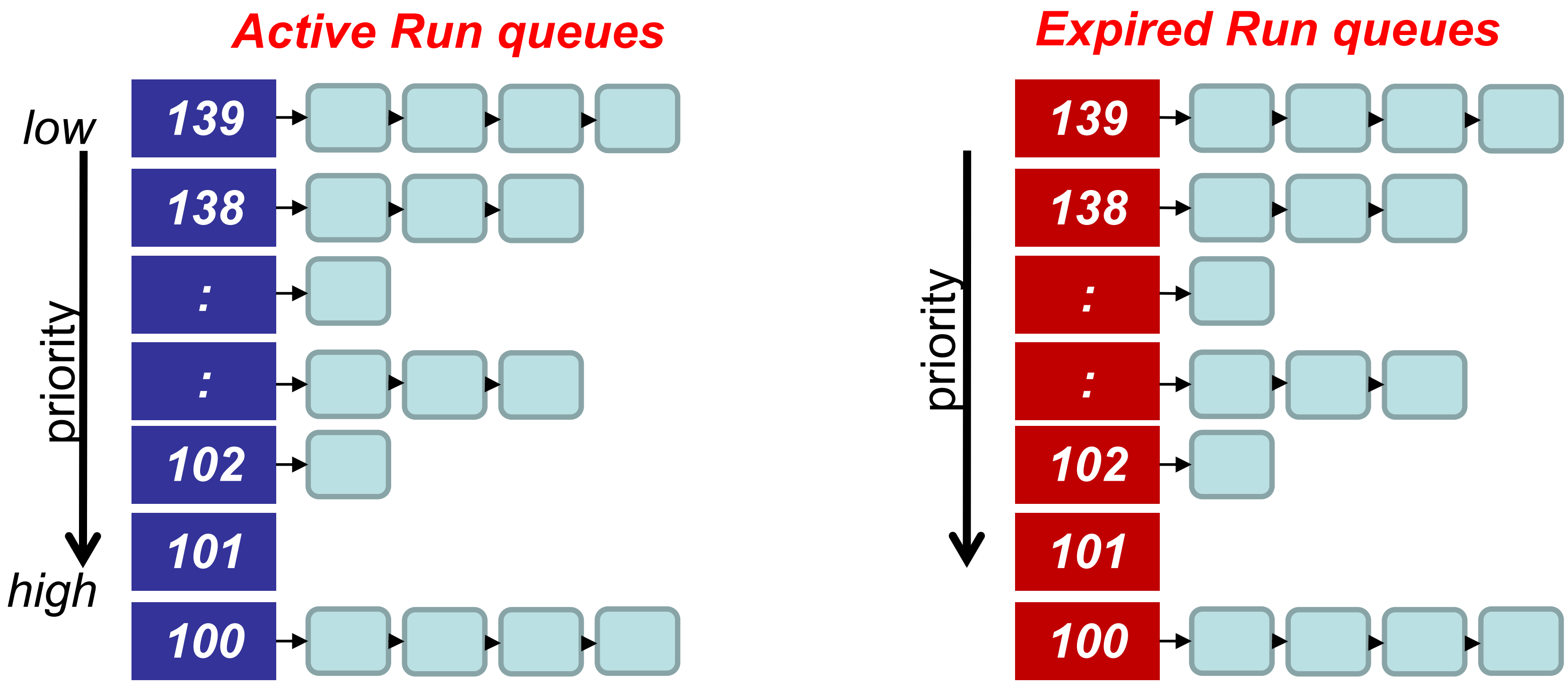
- O(n), when n is the number of runnable processes … not scalable!!
  - Scalability issues observed when Java was introduced (JVM spawns many tasks)
- Used a global run-queue in SMP systems
  - Again, not scalable!!

O(1) scheduler

- Constant time required to pick the next process to execute
  - easily scales to large number of processes
- Processes divided into 2 types
  - Real time
    - Priorities from 0 to 99
  - Normal processes
    - Interactive
    - Batch
    - Priorities from 100 to 139 (100 highest, 139 lowest priority)

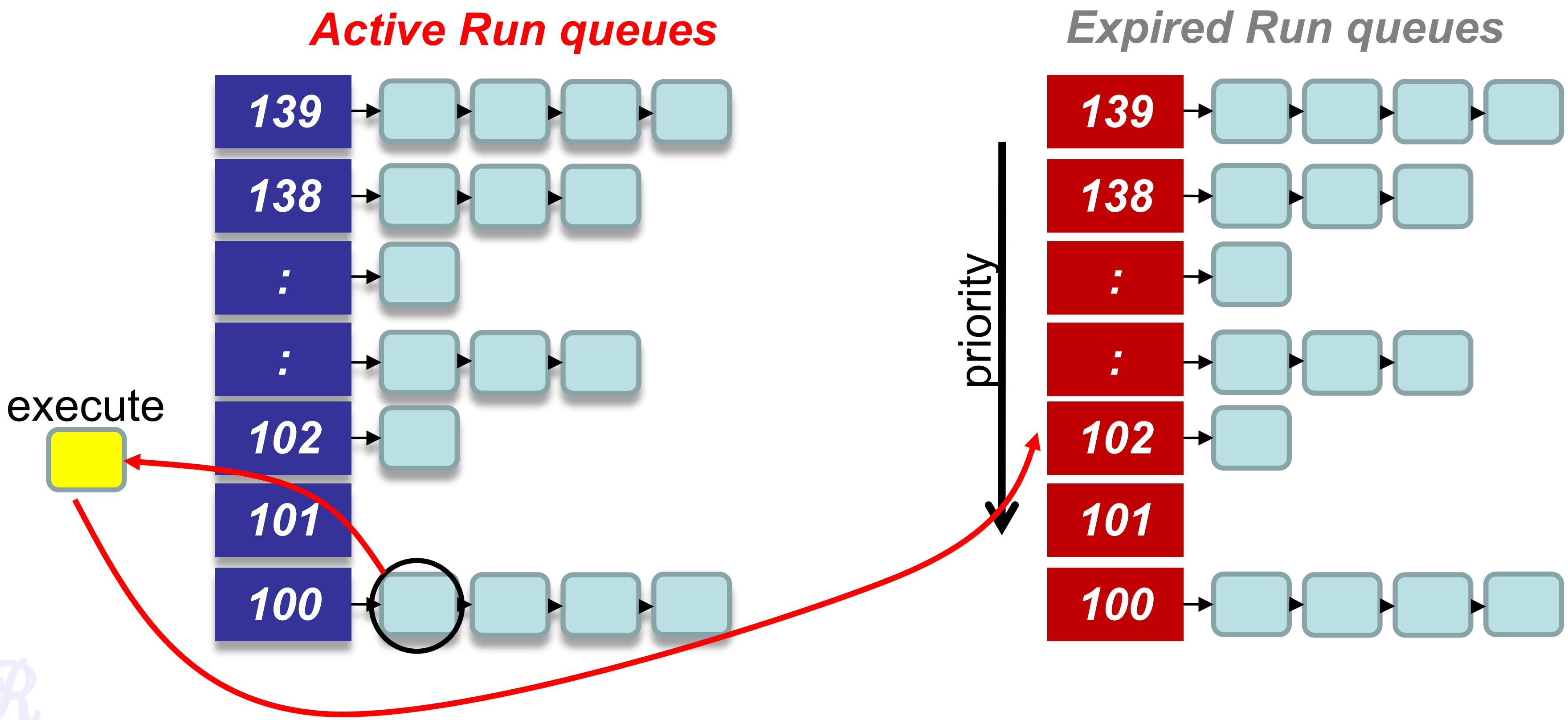
Scheduling Normal Processes

- Two ready queues in each CPU
  - Each queue has 40 priority classes (100 – 139)
  - 100 has highest priority, 139 has lowest priority



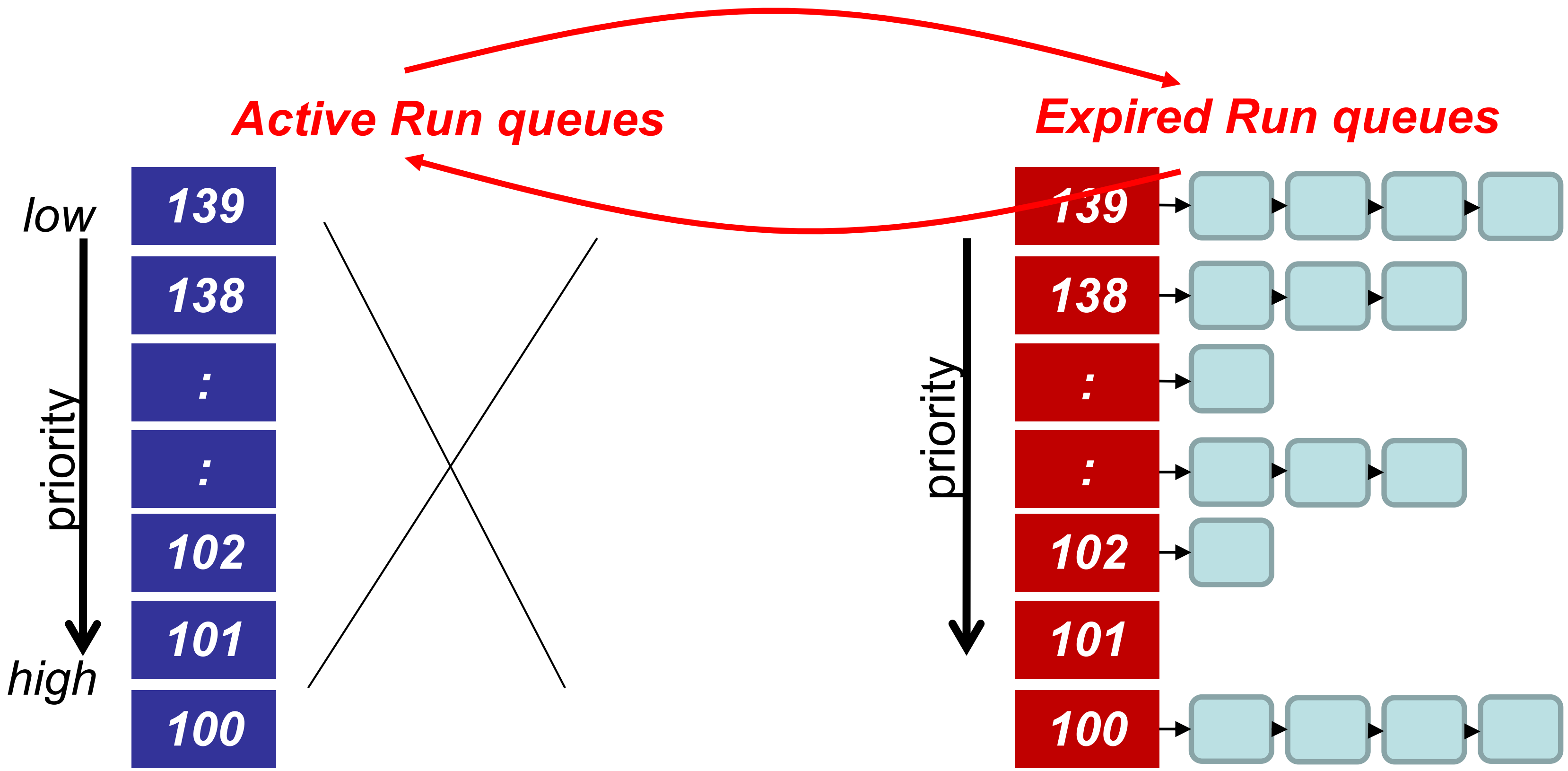
The Scheduling Policy

- Pick the first task from the lowest numbered run queue
- When done put task in the appropriate queue in the expired run queue



The Scheduling Policy

- Once active run queues are complete
  - Make expired run queues active and vice versa



contant time?

- There are 2 steps in the scheduling
  1. Find the lowest numbered queue with at least 1 task
  2. Choose the first task from that queue

- step 2 is obviously constant time
- Is step 1 contant time?
  - Store bitmap of run queues with non-zero entries
  - Use special instruction ‘*find-first-bit-set*’
    - *bsfl* on intel

More on Priorities

- 0 to 99 meant for real time processes
- 100 is the highest priority for a normal process
- 139 is the lowest priority
- Static Priorities
  - 120 is the base priority (default)
  - **nice :** command line to change default priority of a process

    $nice –n N ./a.out
  - N is a value from +19 to -20;
    - most selfish ‘-20’; (I want to go first)
    - most generous ‘+19’; (I will go last)

Based on a heuristic

Dynamic Priority

- To distinguish between batch and interactive processes
- Uses a ‘bonus’, which changes based on a heuristic

dynamic priority = MAX(100, MIN(static priority – bonus + 5), 139))

Has a value between 0 and 10

If bonus < 5, implies less interaction with the user
thus more of a CPU bound process.
The dynamic priority is therefore decreased (toward 139)

If bonus > 5, implies more interaction with the user
thus more of an interactive process.
The dynamic priority is increased (toward 100).

Dynamic Priority (setting the bonus)

- To distinguish between batch and interactive processes
- Based on average sleep time
  - An I/O bound process will sleep more therefore should get a higher priority
  - A CPU bound process will sleep less, therefore should get lower priority

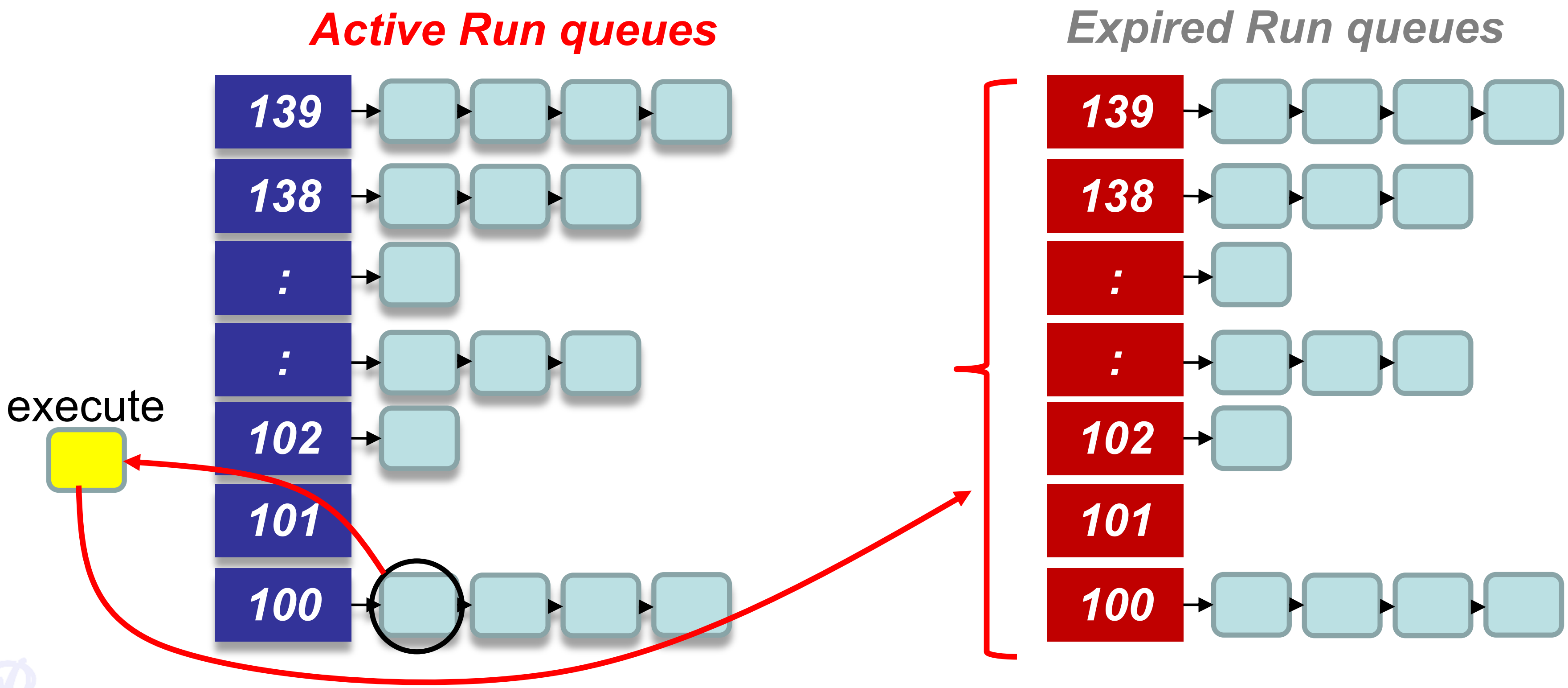
dynamic priority = MAX(100, MIN(static priority – bonus + 5), 139))

heuristic

Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9
1 second	10

Dynamic Priority and Run Queues

- Dynamic priority used to determine which run queue to put the task
- No matter how ‘nice’ you are, you still need to wait on run queues --- prevents starvation



Setting the Timeslice

- Interactive processes have high priorities.
  - But likely to not complete their timeslice
  - Give it the largest timeslice to ensure that it completes its burst without being preempted. More heuristics

If priority < 120
    time slice = (140 – priority) * 20   milliseconds
else
    time slice = (140 – priority) * 5   milliseconds

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	19	5 ms

Summarizing the O(1) Scheduler

- **Queues:** Multi level feed back queues with 40 priority classes
- **Base Priority:** Base priority set to 120 by default; modifiable by users using nice.
- **Dynamic Priority:** Dynamic priority set by heuristics based on process' sleep time
- **Dynamic timeslices:** Time slice interval for each process is set based on the dynamic priority
- **Starvation:** is dealt with by the two queues

Limitations of O(1) Scheduler

- Too complex heuristics to distinguish between interactive and non-interactive processes
- Dependence between timeslice and priority
- Priority and timeslice values not uniform

Completely Fair Scheduling (CFS)

- The Linux scheduler since 2.6.23
- By Ingo Molnar
  - based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas.
  - Incorporated in the Linux kernel since 2007
- No heuristics.
- Elegant handling of I/O and CPU bound processes.

Completely Fair Scheduling (CFS)

Ideal Fair Scheduling

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

execution with respect to time

Ideal Fair Scheduling

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

Each process gets
4/4 = 1ms of the processor time

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

execution with respect to time

Ideal Fair Scheduling

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

Each process gets
4/2 = 2ms of the processor time

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

execution with respect to time

Ideal Fair Scheduling

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

The single process gets the entire 4ms of the processor time

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms period

execution with respect to time

Virtual Runtimes
(keeping track of execution time)

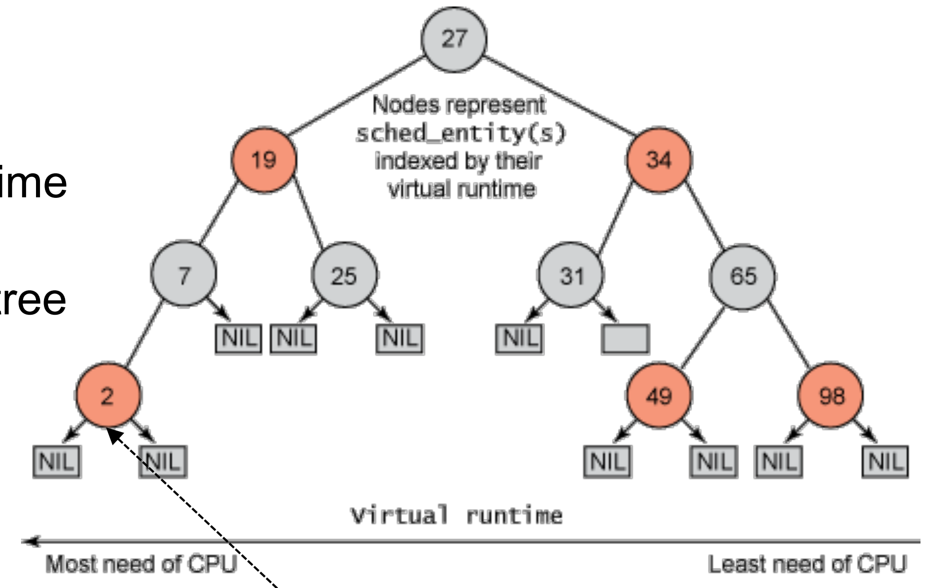
- With each runnable process is included a virtual runtime (vruntime)
  - At every scheduling point, if process has run for t ms, then (vruntime += t)
  - vruntime for a process therefore monotonically increases

The CFS Idea

- When timer interrupt occurs
  - Choose the task with the lowest vruntime (min_vruntime)
  - Compute its dynamic timeslice
  - Program the high resolution timer with this timeslice
- The process begins to execute in the CPU
- When interrupt occurs again
  - Context switch if there is another task with a smaller runtime

Picking the Next Task to Run

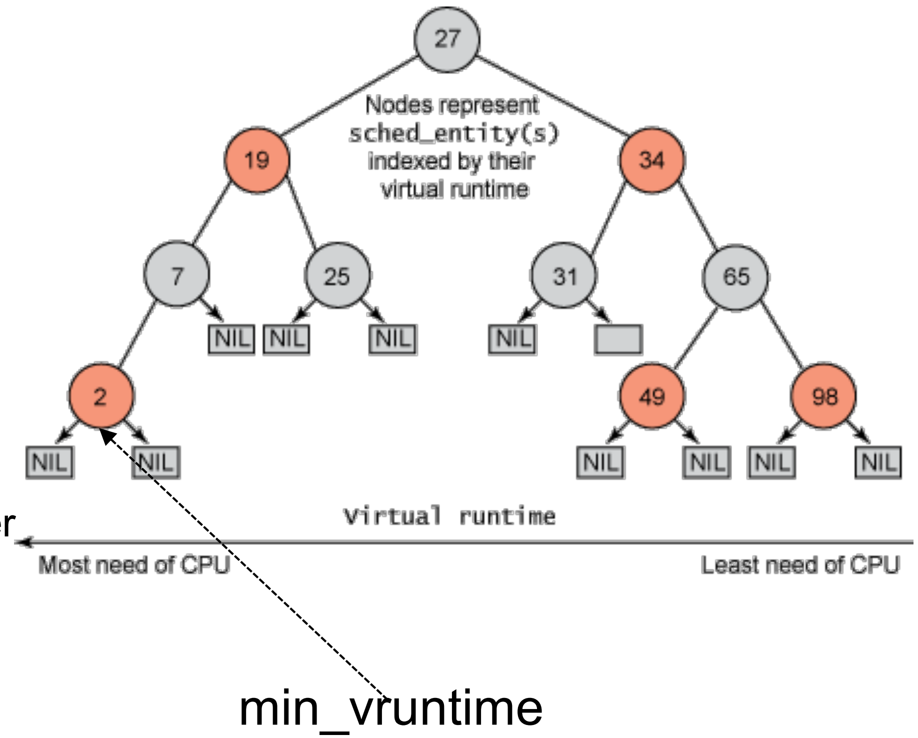
- CFS uses a red-black tree.
  - Each node in the tree represents a runnable task
  - Nodes ordered according to their vruntime
  - Nodes on the left have lower vruntime compared to nodes on the right of the tree
  - The left most node is the task with the least vruntime
    - This is cached in min_vruntime



min_vruntime

Picking the Next Task to Run

- At a context switch,
  - Pick the left most node of the tree
    - This has the lowest runtime.
    - It is cached in min_vruntime. Therefore accessed in O(1)
  - If the previous process is runnable, it is inserted into the tree depending on its new vruntime. Done in O(log(n))
    - Tasks move from left to right of tree after its execution completes… starvation avoided



Why Red Black Tree?

- Self Balancing
  - No path in the tree will be twice as long as any other path

- All operations are O(log n)
  - Thus inserting / deleting tasks from the tree is quick and efficient

Priorities and CFS

- Priority (due to nice values) used to weigh the vruntime

- if process has run for t ms, then

  vruntime += t * (weight based on nice of process)

- A lower priority implies time moves at a faster rate compared to that of a high priority task

I/O and CPU bound processes

- What we need,
  - I/O bound should get higher priority and get a longer time to execute compared to CPU bound
  - CFS achieves this efficiently
    - I/O bound processes have small CPU bursts therefore will have a low vruntime. They would appear towards the left of the tree…. Thus are given higher priorities
    - I/O bound processes will typically have larger time slices, because they have smaller vruntime

New Process

- Gets added to the RB-tree
- Starts with an initial value of min_vruntime..
- This ensures that it gets to execute quickly

Thank You